논문 2011-1-25

하나의 속성을 사용하는 속성 문법 작성의 일반화에 대한 연구

A Research on the Generalization of the Construction of an Attribute Grammar Using One Attribute

정용주*

Yongju Chung

요 약 속성문법은 각 구문규칙에 의미론을 추가한 문법체계이다. 이 속성문법은 두 가지의 속성을 사용하는데 기존 문법에 이 두 가지 속성들로 파싱 과정을 이해하며 추가적으로 작성하여야 하기에 구성의 어려움이 있다. 그래서 본 논문에서는 속성문법을 보다 쉽게 작성할 수 있도록 세 가지 용어들을 정의하며 속성과 속성문법의 분석을 하였고 그래서 경우에 따라서는 하나의 속성만으로도 속성문법을 작성할 수 있다는 가능성을 보여준다.

Abstract An attribute grammar is a set of semantic rules added to the syntax rules. This attribute grammar uses two attributes. It is difficult to write by its additional rules to the existing syntax rules with two attributes understanding the parsing steps. So this paper analyses attributes and an attribute grammar to construct the attribute grammar easily proposing three definitions and shows a possibility that an attribute grammar can be written with only one attribute in some cases.

Key Word : Nonterminal, Attribute Grammar, Inherited Attribute, Synthesized Attribute

I. 서론

속성문법은 정적 의미론을 위한 규칙들로 기존 문법에 의미론만 추가하여 언어 개발에 많이 사용하는 방법의 하나이다[1]. 이 문법에는 의미를 상향식으로 의미를 전달하는 합성속성(synthesized attribute)과 하향식으로 전달하는 상속속성(inherited attribute) 두 가지를 사용하여 파싱 단계에서 구문 안의 특정 부분의 속성이 맞는지 확인한다[2][3].

이 속성문법은 파싱되는 단계 단계에 맞추어 합성, 상속속성 두 속성으로 규칙들을 작성하여 술어함수와 함께 의미를 확인한다. 그리고 이것은 문법이 틀리지 않으면 어떠한 형태의 문법이든 사용할 수 있는 큰 장점이 있다. 즉, LR 방식을 위한 어떠한 형태의 문법이든 올바르게 파싱이 되면 그 문법에 이 속성문법을 작성할 수 있는 좋은 방법이다. 그러나 여기에는 어려운 점도 있다[3]. 우선 작성의 어려움이 있고 또한 이렇게 제작된 속성문법을 이해하는 어려움이 있다. 속성문법의 작성이 어려운 이유는 속성문법 작성자는 스택을 포함 파싱되는 과정 단계를 파악하고 있어야 하고 또한 각 상황에 맞는 하향식 속성과 상향식 속성들의 규칙을 작성하여야 하기 때문이다. 이것의 작성 방법은 체계적인 접근 방법이 없고 문법 구성에 따라 다르므로 아무래도 작성자의 경험에 많이 의존한다. 그리고 또 다른 문제로는 두 가지 속성을 사용하므로 전체적인 규모가 커진다는 문제가 있다. 그래서 속성 문법 구성을 쉽게 혹은 단순하게 할 수 있으면 좋다. 이에 대한 연구는 속성문법 발표 초기에 속성문법 자체

*정회원, 단국대학교 컴퓨터학부
접수일자: 2010.12.31 수정일자: 2011.1.27
게재확정일자: 2011.2.11

를 위한 응용이나 활용, 제약에 대한 발표는[4][5][6][7][8] 있었으나 구성을 단순화하는 연구는 많지 않은 것으로 보인다. 그래서 본 논문에서는 이러한 단순한 속성문법 구성을 위해서 속성과 문법에 관하여 연구하였다. 어떠한 경우에 두 가지 속성들이 모두 필요하고 어떠한 경우에 하나의 속성만 있어도 속성문법의 구성이 가능한지를 나름대로 분석하였다. 속성을 하나만 사용하는 경우는 하나의 속성값, 상향식 속성값 만으로 속성문법 작성이 가능하므로 속성문법 구성 자체는 보다 간편하게 구성될 수 있다. 또한 그로인해 전체 규모도 축소할 수 있다.

본 논문에서의 분석은 어디까지나 그 가능성이다. 실제적인 언어에 적응하기까지는 더 많은 연구가 필요할 것이다. 그러나 속성문법을 반드시 두 개의 속성으로 작성하여야 한다는 고정 관념을 바꿀 수는 있을 것으로 보인다. 또한 속성문법이 구문 지향 편집 시스템, 자연 언어 처리 시스템 등에 많이 사용이 되고[9][10] 이 중 구문 지향 편집 시스템에서는 다양한 속성들이 사용되지 않을 것으로 추측되는데 이러한 시스템이라면 본 논문에서의 방식도 의미가 있다고 판단된다. 그리고 본 논문에서의 문장들의 문법 규칙들은 Chomsky에 의한 type2 문법을 전제로 한다.

II. 두 개의 속성이 필요한 이유

LR은 왼쪽에서 우측으로 문장을 읽으며 우파스를 출력한다. 이 때 터미널을 입력받아서 생성규칙의 우측 부분을 좌측 부분으로 대체하며 점차 시작 기호까지 상향식으로 올라가는 방식이다[11][12]. 즉, 문법 작성을 시작 기호부터 시작하여 점차 아래로 터미널을 생성하는 규칙으로 작성하였다면 LR 파싱은 아래에서 위로 파싱하는 방법이다. 그래서 속성이 필요한 변수에 대해서는 심볼 테이블에서 그 값을 찾아서 넌터미널로 대치하며 상향으로 올라간다. 따라서 파싱하는 과정에서 어떠한 변수의 속성값이나 아니면 어떠한 정보라도 상향으로 전달하는 경우는 큰 문제가 없다. LR의 기본 방향과 일치하므로 단순히 그 정보 혹은 속성값을 상향으로 전달하면 된다.

그러나 어떠한 정보를 하향으로 전달하여야 하는 경우는 어려움이 생긴다. LR의 기본 방향과 반대이기 때문이다. 대표적인 예가 수식의 속성을 결정하여야 하는 경우이다. 수식은 수식 자체가 문장이 되는 것은 아니고 어떠한 문장 안의 일부로 사용된다. 예를 들어 배정문, if문, for문 등 문장 안에서 그 일부로 사용된다. 그리고 수식은 그러한 문장들 안에서 속성이 미리 결정이 된다. 예를 들어 배정문으로 y = x1+... 이고 y의 속성이 int 라면 수식의 속성은 int이고 if문이나 while문 안에서는 그 수식의 속성이 논리형으로 미리 정해진다. 즉, 수식은 LR 방식에서 왼쪽에서 오른쪽으로 읽고 파싱하는 방식이므로 그 자료형이 미리 정해진 후에 파싱이 시작된다. 수식과 같은 경우, 미리 정해진 속성으로 파싱을 시작하면서 속성 확인을 하여야 한다면 LR의 경우 문제가 된다. 예를 들어 수식을 나타내는 넌터미널을 <expr>이라고 하면 이 <expr>을 확장하며-LR의 경우 shift- LR의 기본 방향과는 반대의 방향으로 미리 결정된 속성값, 정보를 가지고 하향으로 내려가야 하기 때문이다. 그래서 또 하나의 속성이, 즉 상속속성이 필요하다. 즉, 상속속성은, 문장 안에서의 어느 부분이 파싱이 끝난 후 자연스럽게 그 부분의 속성 확인을 하는 상황이 아니고, 파싱 전에 확인할 속성이 미리 정해져서 그 정보(속성값)를 파싱을 하며 하향으로 전달하여야 하는 경우에 필요하다. 그러나 반드시 필요한 것은 아니다. 가령 그 미리 정해진 정보가 어떠한 방법에 의하여 유지하거나 전달하지 않아도 되는 상황이면 이 상속속성은 불필요하게 된다. 그 어떠한 방법이라는 것은 여러 가지가 있을 수 있겠으나 본 논문에서는 속성과 속성문법에 관하여 연구하여 보았다.

어떠한 경우에 속성문법의 속성이 하나 필요하고 어떠한 경우에 두 개가 필요한지는 문법의 구성 방법에 따라서 다를 수 있다. 이러한 것들을 개념적으로 이해하고 정리하기 위해서 본 논문에서는 문법 규칙에 대하여 세 가지 용어들, “자료형 확인 요구 규칙”, “자료형 결정 규칙”, “요구된 자료형 결정 규칙”들을 기술한다.

첫째 “자료형 확인 요구 규칙”이란 문법 규칙들 중에서 자료형 확인을 요구하는 규칙이다. 예를 들어 if문의 경우 if 문을 문법으로 표기하면 if ′(′<l-expr>‘)’ ... 으로 될 때 바로 문법에서 이 규칙을 말한다. 둘째 “자료형 결정 규칙”이란 자료형을 찾거나 계산 혹은 결정하는 규칙 부분으로, 예를 들어 심볼 테이블에서 변수의 자료형을 찾는다든지 혹은 x1 + x2 등 수식에서 x1, x2 자료형에 따라 수식의 전체 자료형을 결정하여야 하는 규칙들을 의미한다. 그리고 “요구된 자료형 결정 규칙”이란 요구된 자료형이 최종적으로 파서에 의해서 결정된 자료형과 일치하는지 확인하는 규칙을 의미한다. 예를 들어 앞의 if

문에서 <l-expr>은 수식의 속성값으로 논리형을 요구하게 되고 이 <l-expr>은 다른 넌터미널들로 대치되는데 (혹은 계속적으로 대치되어) 최종적으로 자료형 결정 부분과 만나게 되는데 바로 이 부분을 말한다. 속성문법에서 보통 이 부분에 술어함수가 있는데 이 문법규칙이다.

이 세 가지 규칙들에서 자료형 확인 요구 규칙은 정해져 있다. 예를 들면 배정문, if문 등은 그 규칙에서 속성확인을 요구한다. 그리고 이것은 이 세 종류의 규칙들 중에서 속성 확인을 할 때 이 부분부터 파싱이 시작된다. 다음 단계는 자료형 결정 규칙이다. 보통은 여러 개의 결정 규칙들을 거치며 다음 단계인 요구된 자료형 결정 규칙을 사용하게 된다. 이 요구된 자료형 결정 규칙은 정해져 있는 것은 아니다. 앞서 기술하였지만 요구된 자료형을 최종적으로 확인하는 술어함수를 놓는 규칙이 이 요구된 자료형 결정 규칙이 된다. 이와 같은 용어들로 다음과 같은 정리가 가능하다.

정리 1) 일반적인 경우, 속성문법에서 어느 한 넌터미널의 속성을 확인할 때 자료형 확인 요구 규칙과 그 넌터미널로부터 파생되어 생긴 요구된 자료형 결정 규칙이 서로 분리되어 있는 경우 두 속성, 상속속성과 합성속성들이 모두 필요하다.

증명) 자료형 확인 요구 규칙을

A → $A_0 A_1 ... A_i .. A_{k1}$ 이라하고 이중 A_i가 자료형의 확인을 요구하고 또 이로부터 파생(replace)되어 요구된 자료형을 결정하는 요구된 자료형 결정 규칙을 M → $M_0 M_1 ... M_{km}$ 이라 하자.

M을 위한 규칙은 요구된 자료형 결정 규칙이므로 넌터미널 A부터 M까지는 요구된 자료형의 정보를 전달하여야 한다. 따라서 상속속성이 필요하다. 또한 M을 위한 규칙에서 M_0 ... M_{km} 넌터미널들의 최종 자료형 결정을 위해서 최소한 한 번 이상은 심볼 테이블로부터 시작된 상향으로 전달된 정보 전달이 있었다. 따라서 합성속성이 필요하다. 따라서 두 속성들은 모두 필요하다.

이 정리는, 배정문과 같이, 넌터미널 A 를 위한 규칙 안에 두 가지 이상의 넌터미널들이 관련되어 있어도 적용된다. 왜냐하면 각각 따로따로 적용하면 되기 때문이다. 예를 들어 정리의 A_i가 <id>이고 A_{i+1}이 '=', A_{i+2}가 수식을 위한 <expr> 넌터미널이라면 <id>의 자료형이 심볼 테이블로부터 먼저 결정이 되고 이것이 float이라면 A_{i+2}는 float 자료형을 가지고 하향식으로 내려간다.(이 경우의 A_{i+2}는 정리의 A_i)

III. 속성값의 종류와 속성문법 작성

1. 하나의 속성값만 사용하는 경우

앞의 2절에서 세 가지 용어들을 기술하였다. 이들 중에서 "자료형 확인 요구 규칙"과 "요구된 자료형 결정 규칙"을 일치시킬 수 있다고 가정하자. 여기서 두 규칙을 일치시킬 수 있다는 것은 최종 자료형 요구 규칙을, 최종 술어함수를 하나의 속성만으로 속성문법을 작성하기 위해서 단순히 자료형 확인 요구 규칙에 놓는다는 것을 의미하지는 않을 것이다. 문법 구성으로 아니면 어떠한 방식이든 두 규칙들을 일치시켜도 속성확인이 되어야 한다는 것을 전제로 한다. 그러나 사용하는 속성값이 하나인 경우 그리고 그것을 확인하는 경우는 두 규칙을 단순히 일치시켜도 하나의 속성만 사용하여도 속성 확인은 가능하다.

속성문법은 기존 문법에 추가하여 작성한다고 하였다. 이것은 기존 문법이 옳게 파싱된다는 전제로 한다. 그래서 이것은 속성의 종류가 하나이고 여러 변수들의 속성이 그 종류의 속성값을 가지고 있는지 아닌지만을 확인한다면 단지 그 변수들이 그 속성을 가지고 있는지만을 확인해도 된다는 것을 의미한다. 왜냐하면 변수가 있어야 할 곳에 명령어나 다른 것이 있는 것은 구문문법 에러이지 속성 에러는 아니기 때문이다. 따라서 속성문법을 작성할 때 해당 변수들이 심볼 테이블로부터 속성 값을 받아서 단지 상향으로 올려서 확인하면 된다. 예를 들어 넌터미널 A_i가 문장의 한 부분을 이루는 것으로 이 A_i 안의 변수들의 속성을 확인하려 한다면 A_i로부터 파생되어 생성되는 변수들에서부터 속성값을 받아서 중간 중간 각 넌터미널들에 그 값을 계속 전달하여 이 A_i에서 확인하면 된다. 심볼 테이블에서 직접 확인하는 경우는, 즉 변수들이 심볼 테이블에서 속성값을 받을 때 그 자리에서 속성 확인을 하는 것은 문제가 생길 수 있다. 왜냐하면 변수가 또 다른 어떠한 문장에서 사용될 지 예측할 수 없기 때문이다. 이러한 관계를 정리하면 다음과 같다.

정리2) 자료형 확인 요구 규칙을

A → $A_0 A_1 ... A_i .. A_{k1}$ 이라하고 이중 A_i가 자료형의 확인을 요구하는데 A_i는 오직 하나의 속성만을 사용하며 그 자료형의 확인을 요구한다면 속성문법의 최종 술어함수를 이 규칙에 놓고, A_i에 관련된 변수들의 속성값을 심볼 테이블로부타 찾아서 단지 상향으로 전달하여 결정하면 된다.

정리2를 따르면 어떠한 언어에서 변수의 속성으로는 논리형만이 잇고 수식으로는 논리식만이 있으면 속성 확인은 하나의 속성만으로도 가능하다. 왜냐하면 각 변수를 읽을 때 논리형인지만 확인하면 되기 때문이다.

표1은 단순히 논리연산자와 논리변수만 있다고 가정하고 if문에 대해서 작성한 구문문법과 속성문법이다.

예를 들어 if(a and b) ... 의 경우 a는 규칙7에서 심볼 테이블로부터 속성값을 찾고 규칙5에서 'logical'임을 확인한다. b도 같은 방식으로 'logical'임을 확인하고 규칙4에서 <l-term>으로 속성값을 전달한다. 이것은 계속 상향으로 전달되어 규칙1의 if문에서 확인한다.

표 1. 논리형 변수와 논리 연산자만을 사용하는 논리식을 위한 속성 문법

Table1. Attribute grammar for a logical expression

1. <if> →'if' '('<expr>') <stmt> (술어함수)<expr>.type='logical'
2. <expr>[1] → <expr>[2]'or'<l-term> (속성규칙) <expr>[1].type ← <l-term>.type
3. <expr> → <l-term> (속성규칙)<expr>.type ← <l-term>.type (술어함수) <expr>.type = 'logical'
4.<l-term>[1]→<l-term>[2]'and'<l-factor> (속성규칙)<l-term>[1].type ← <l-factor>.type
5. <l-term> → <l-factor> (속성규칙)<l-term>.type ← <l-factor>.type (술어함수) <l-term>.type = 'logical'
6. <l-factor> → (<logical>) (속성규칙)<l-factor>.type ← <logical>.type
7. <l-factor> → <id> (속성규칙)<l-factor>.type←lookup(<id>.string)

2. 속성값의 종류가 여러 개인 경우

문장 안의 어느 부분의 속성값이 앞 절의 경우와 같이 하나의 속성값 종류만 있는 것이 아니고 두 개 이상이 존재하는 경우이다. 예를 들면 프로그래밍 언어의 수식에는 int, real, 논리형 등 여러 가지의 속성값들이 존재한다. 이러한 경우는 속성문법 작성을 할 때 두 가지 속성들을 모두 사용하는 것으로 되어 있다.

속성값의 종류가 여러 개인 경우 그 속성값들의 종류를 속성값의 특성으로 나눌 수 있다. 첫째는 속성값의 종류가 서로 관련이 없는 경우이다. 즉, 속성값의 종류가 여러 개라도 어떤 한 변수가 가질 수 있는 속성값의 종류는 오로지 하나인 경우이다. 예를 들면 파일 이름, 객체 이름, port 번호 등 속성값들 사이에 어떠한 관계도 없는 경우이다. 이런 경우, 즉 문장 안의 어느 한 변수에게 이러한 속성값 종류 중에 하나만 허락이 되는 경우이다. 이 경우는 오직 그것만 확인하면 된다. 따라서 한 문장 안에 이러한 것들이 함께 사용되는 경우가 있다 하여도 각각 따로따로 정리2의 방식을 적용하면 될 것이다. 즉 속성을 하나만 사용하면 단순히 심볼 테이블에서 속성값을 찾아서 상향으로 전달하여 자료형 요구 규칙에서 각각 확인하면 된다.

속성값 종류들 사이에 이질적이지 않고 어떠한 관계가 존재하는 경우이다. 이것은 어떠한 넌터미널이 하나의 속성값만을 가질 수 있는 것이 아니고 여러개 중에 하나를 가질 수 있는 상황이다. 예를 들면 수식은 수식의 넌터미널이 <expr>이라면 int, real, 논리형들을 가질 수 있다. 그리고 이들 자료형들은 서로 간에 관계가 있다. 즉 보통의 프로그래밍 언어에서는 수식에서 int와 real이 사용되면 그 수식은 real로 결정된다. 이러한 경우 속성문법을 하나의 속성으로 혹은 아니면 두 개의 속성으로 작성할 수 있다고 단정하기는 어렵다. 여러 언어론 교재나 컴파일러 교재에서는 수식의 속성문법 작성을 두 개의 속성, 상속속성과 합성속성 모두를 사용하며 작성한다. 이것은 문법을 자유롭게 작성할 수 있도록 하고 속성문법 작성을 다양하게 할 수 있다. 그러나 문법 구성에 따라서는 하나의 속성만으로 속성문법을 작성할 수 있다.

어떠한 변수가 서로 이질적이지 않은 여러 속성값들 중에서 하나를 가질 수 있다면 그 속성값들 사이에는 어떠한 관계가 있을 수 있다. 예를 들면 수식에서 변수는 int와 real을 가질 수 있는데 이 둘은 정확도에서 차이가 있고 real이 int보다 더 정확하다. 속성값들 사이에 어떠

한 관계가 존재하면 속성문법 작성을 하나의 속성만으로 가능하다.

표2는 int와 real만 있는 수식을 위한 문법과 속성문법이다. 이 방식은 두 속성 int와 real 간의 관계를 고려하여 작성한 것이다. 그리고 속성값은 단순히 상향으로 전달하지만 int와 real이 함께 있는 상황에서는 선택을 하였다. 이와 같이 속성값의 종류가 두 개 이상인 경우도 하나의 속성만으로 단순하게 속성문법을 작성할 수 있다.

표 2. int와 real을 사용하는 수식의 속성문법
Table 2. Attribute grammar of an expression using 'int' & 'real'

규칙1: <expr>[1] → <expr>[2]+<term>
(속성문법)
<expr[1]>.synthesized ←
if(<expr>[2].synthesized=int) and
(<term>[3].synthesized=int)
then int
else real
end if

규칙2:<expr> →<term> (속성문법) <expr>.synthesized ←<term>.synthesized

규칙3:<term>[1] → <term>[2]*<factor>
(속성문법)
<term[1]>.synthesized ←
if(<term>[2].synthesized=int) and
(<factor>.synthesized=int)
then int
else real
end if

규칙4:<term> → <factor> (속성문법) <term>.synthesized← <factor>.synthesized
규칙5:<factor> → <id> (속성문법) <factor>.synthesized←<id>.synthesized
규칙6:<factor> → (<expr>) (속성문법) <factor>.synthesized←<expr>.synthesized

표2를 사용하여 정리1에 따라서 최종 수식이 사용된 문장에 최종 술어함수를 넣으면 하나의 속성만으로 속성문법의 작성이 가능하다. 예를 들어 배정문의 문법 규칙을 <assign> → <id> '=' <expr> 이라 하면 바로 이곳에 넣을 술어함수는 다음과 같다.

<id>.synthesized=<expr>.synthesized

<id>.synthesized은 심볼 테이블에서 찾아진다. 따라서 하나의 속성만으로 속성문법이 가능하다.

IV. 결론

속성문법은 Kunth의 발표 이후 그 연구가 거의 없는 실정이다. 그러나 그 구성은 여전히 어려움으로 남아 있다. 그래서 본 논문은 속성문법과 속성에 관하여 그리고 하나의 속성만을 사용할 수 있는 경우에 관하여 연구하였다. 이를 위해서 몇 가지 용어들을 소개하며 속성문법에서 속성을 반드시 두 개를 사용하지 않고도 속성문법을 작성할 수 있다고 제안하며 나름대로 증명하려 하였다. 이것은 속성문법의 구성의 어려움을 어느 정도 해결할 수 있는 가능성을 보여준다. 그러나 3절에서와 같이 속성값의 종류가 여러 개인 경우는 여전히 어려움이 남는다. 즉, 자료형 요구 규칙과 자료형 결정 규칙을 일치시키기 위해서는 문법 구성이나 어떠한 방법을 택하든 하향으로 전달할 추가적인 정보가 없어야 하는데 이러한 문법 구성이나 방법은 어려운 문제이다.

참고 문헌

[1] Knuth, D. E., "Semantics of Context-Free Languages", *Mathematical Systems Theory*, Vol. 2, pp. 127-146, 1968

[2] Meyer, B. *Introduction to the Theory of programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1990

[3] Robert W. Sebesta, *Concepts of Programming Languages*, 7th ed., Pesrson Education, 2006

[4] Koster, C. H. A. "Affix Grammars", in *Peck*, pp.

95-109, 1971

[5] Bochmann, G. V., "Semantic Evaluation from Left to Right", *CACM* Vol. 19 pp. 55-62, 1976

[6] Koskimies, K. and K. J. Raiha, "Modelling of Space-efficient One-pass Translation Using Attribute Grammars", *Software - Practice and Experience* 13. 1983

[7] Reps, T. W., *Generating Language Environment*, MIT Press, 1984

[8] Lewis, P. M., D. J. Rosenkrantz and R. E. Stearns, "Attributed Translations" *J. of Computer and System Sciences* Vol. 9, pp. 279-307. 1974

[9] Farrow, R. "Linguist 86: Yet Another Translator Writing System Based on Attribute Grammar", ACM SIGPLAN Notices, Vol. 17, No.6, pp.160-171, 1982

[10] N. Correa, R. C. Berwick etc. "Empty Categories, Chain Binding and Parsing", pp.83-121, Kluwer Academic Publisher, New York, 1992

[11] Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Vol. 1:Parsing, Prentice-Hall, 1972

[12] Alfred V. Aho, etc., *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986

저자 소개

정 용 주(정회원)

- 1977년-1981년 : 서울대학교 자연대 계산통계학과졸
- 1981년-1983년 : KAIST 전산학과 졸
- 1983년-1984년 : 단국대학교 통계문제연구소 연구원
- 1984년 : 단국대학교 공학대 컴퓨터학부 교수
- 1995년-1998년 : RPI 박사과정, 신병으로 휴학후 다시 단국대 복직.